MITRE eCTF 2025

UIUC Design Document

Last updated	February 26, 2025
School	University of Illinois Urbana-Champaign
Team	SIGPwny
Advisor	Kirill Levchenko
Members	Minh Duong, Jake Mayer, Nikhil Date, Suchit Batpala, Akhil Bharanidhar, Stephen Cao, Sagnik Chakraborty, Sanay Doshi, Yotam Dubiner, Timothy Fong, Serkan Gozel, Yash Gupta, Adish Jain, Peter Karlos, Swetha Karthikeyan, Adarsh Krishnan, Ananth Madan, Mason Miao, Safwan Morshed, Julie Oh, Ben Pegg, Jupiter Peng, Phillip Raczka, Liam Ramsey, Neil Rayu, Rohan Seth, Preetesh Shah, Krishnan Shankar, Arpan Swaroop, Saipranav Venkatakrishnan

SIGPwny

Table of Contents

1 Introduction

This document describes UIUC's implementation of a secure satellite TV system for the MITRE eCTF 2025 competition.

1.1 Competition Overview

While the competition does not actually use satellite networks, a simulated network is provided by the competition organizers through the supplied host tools. These host tools simulate the uplink, satellite, and downlink components of a satellite network. These components are out-of-scope for implementation, and teams instead focus on implementing the Encoder and Decoder components, as well as the build system for these components.

- **Encoder**: Transforms input data streams into encoded streams, which is sent to the uplink, then to the satellite. The satellite then transmits one-way to the receiver, which passes the data to the Decoder.
- **Decoder**: Receives encoded streams from the receiver, decodes them, and outputs the original data streams.

The Encoder is meant to run on any standard operating system as its interface is required to be compatible with the host tools written in Python. The Decoder runs on Analog Devices' MAX78000FTHR development board, which features the MAX78000 microcontroller.

1.1.1 Security Requirements

Below are the Security Requirements (SRs) provided by the competition organizers, which are meant to defend against the Attack Scenarios.

- **SR1**: An attacker should not be able to decode TV frames without a Decoder that has a valid, active subscription to that channel.
- **SR2**: The Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for.
- **SR3**: The Decoder should only decode frames with strictly monotonically increasing timestamps.

Due to the embedded systems focus of the competition, we believe that it is necessary to introduce an additional security requirement to defend against hardware attacks:

- **SR4**: The Decoder's operations should be resistant to side-channel analysis, fault injection, and other physical attacks.

1.1.2 Attack Scenarios

The following Attack Scenarios are considered by the competition organizers. Successful attacks against these scenarios result in a flag, which is a string in the form of `ectf{...}`.

- **AS1** - Expired Subscription: Read frames from a channel you have an expired subscription for.
- **AS2** - Pirated Subscription: Read frames from a channel you have a pirated subscription for (i.e. a subscription that was intended for another Decoder).
- **AS3** - No Subscription: Read frames from a channel you have no subscription for.
- **AS4** - Recording Playback: Read frames from a recorded channel you currently have a subscription for, but didn't at the time of the recording.
- **AS5** - Pesky Neighbor: Spoof the signal of the satellite to cause your neighbor's decoder to decode your frames instead.

1.2 Design Overview

To effectively address the Security Requirements and Attack Scenarios, we made the following design decisions.

1.2.1 Rust

Rust is a systems programming language that provides memory safety, thread safety, and other features that make it well-suited for embedded systems development. Rust's strong type system and ownership model helps prevent common programming errors that can lead to security vulnerabilities, such as buffer overflows.

We decided to implement both the Encoder and Decoder in Rust to take advantage of these features. To support the required Python interface for the Encoder, we used PyO3 to generate Python bindings for our Rust code and Maturin to build and precompile wheel files for local installation. The performance of our Encoder is also much improved by using Rust, as it is able to encode frames much faster than a Python implementation.

The MAX78000 microcontroller is a dual-core MCU with an Arm Cortex-M4 and a RISC-V core. We primarily use the CM4 core for our implementation, and leave the RISC-V core disabled. Arm targets are well-supported by Rust; however, the MAX78000 lacks a widely-supported Rust Hardware Abstraction Layer (HAL). We had previously worked with the MAX78000 in Rust before, so we decided to open source our own HAL implementation as the max7800x-hal crate. Our primary hope in publishing our HAL is that more teams will adopt Rust in their own designs.

1.2.2 Authenticated Encryption

We use the Ascon-128 authenticated encryption scheme to protect the confidentiality and integrity of encoded streams and channel subscription updates. Ascon-128 is a lightweight authenticated encryption scheme that is well-suited for embedded systems, along with countermeasures against side-channel attacks.

2 Security

2.1 Cryptography

At a high level, subscriptions are encrypted by authenticated encryption using a device-specific key. We aim to prevent retargeting or forging for a new device even in the event that one device is completely compromised. Additionally, subscriptions cannot be retargeted to a new channel even in the case of a compromised device key as every channel has a channel-specific key. On the other hand, frames are encrypted using a frame-specific key to ensure that retargeted frames are not even decrypted into their correct plaintext.

In order to prevent key reuse, we use a key derivation function (KDF) to derive unique keys for Ascon in various contexts. For our KDF, we use the Keccak Message Authentiation Code (KMAC) construction as defined in NIST SP800-185.

2.2 Primitives

$\text{Ascon128}_{\text{Enc}}(K, P, D, N) \rightarrow (C, T)$
$\text{Ascon128}_{\text{Dec}}(K, C, T, D, N) \rightarrow (P)$
Ascon-128 is an authenticated encryption scheme that takes a key K, a plaintext message P, associated data D, and a public nonce N as input. It outputs a ciphertext C and an authentication tag T. For decryption, it takes a key K, a ciphertext C, authentication tag T, associated data D, and a public nonce N as input. It outputs the plaintext message P.

$\text{KMAC}(K, X, L, S)$
KMAC is Keccak's keyed message authentication code that takes a key K, a message X, and a domain separator S as input. It outputs a tag of length L.

2.3 Symbols

General

D - Decoder ID (4B)
C - Channel ID (4B)
T_{start} - Start timestamp (8B)

T_{end} - End timestamp (8B)
T_{frame} - Frame timestamp (8B)

Deployment Secrets

S_{base_sub} - Base subscription secret (32B)
S_{base_chan} - Base channel secret (32B)
K_{frame} - Frame key (16B)

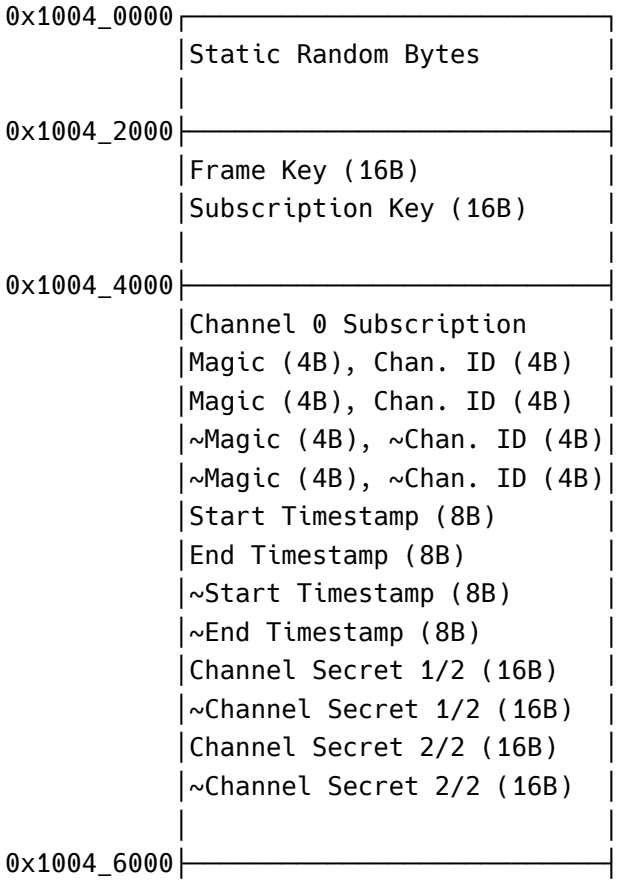
Encoder/Decoder Secrets

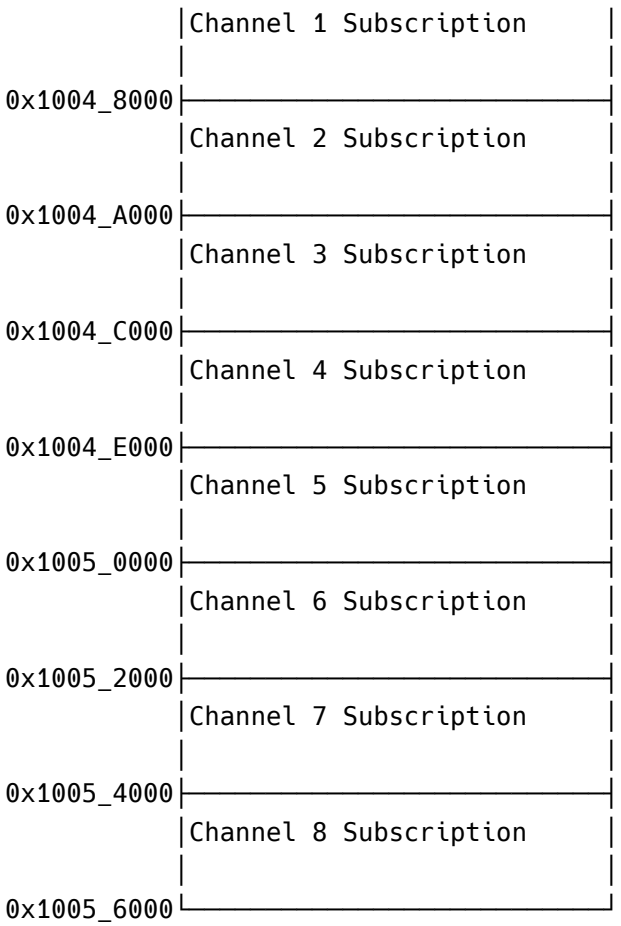
K_{sub} - Subscription key (16B), unique to decoder
S_{chan} - Channel secret (32B), unique to channel
K_{pic} - Picture key (16B), unique to channel and timestamp

Key Derivations

$K_{sub} = \text{KMAC128}(S_{base\_sub}, D, 16\text{B}, \text{"derive\_subscription\_key"})$
$S_{chan} = \text{KMAC256}(S_{base\_chan}, C, 32\text{B}, \text{"derive\_channel\_secret"})$
$K_{pic} = \text{KMAC128}(S_{chan}, (T_{frame} || \sim T_{frame}), 16\text{B}, \text{"derive\_picture\_key"})$

2.4 Secrets in Decoder Flash




Channel 1 Subscription
0x1004_8000
Channel 2 Subscription
0x1004_A000
Channel 3 Subscription
0x1004_C000
Channel 4 Subscription
0x1004_E000
Channel 5 Subscription
0x1005_0000
Channel 6 Subscription
0x1005_2000
Channel 7 Subscription
0x1005_4000
Channel 8 Subscription
0x1005_6000